下载APP

13 | Java JIT编译器（一）：动手修改Graal编译器

2020-07-01 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 19:52 大小 18.21M

你好，我是宫文学。

在前面的 4 讲当中，我们已经解析了 OpenJDK 中的 Java 编译器，它是把 Java 源代码编译成字节码，然后交给 JVM 运行。

用过 Java 的人都知道，在 JVM 中除了可以解释执行字节码以外，还可以通过即时编译（JIT）技术生成机器码来执行程序，这使得 Java 的性能很高，甚至跟 C++ 差不多。反之，如果不能达到很高的性能，一定会大大影响一门语言的流行。

但是，对很多同学来说，对于编译器中后端的了解，还是比较模糊的。比如说，你已经了解了中间代码、优化算法、指令选择等理论概念，**那这些知识在实际的编译器中是如何落地的呢？**

所以从今天开始，我会花 4 讲的时间，来带你了解 Java 的 JIT 编译器的组成部分和工作流程、它的 IR 的设计、一些重要的优化算法，以及生成目标代码的过程等知识点。在这个过程中，你还可以印证关于编译器中后端的一些知识点。

今天这一讲呢，我首先会带你理解 JIT 编译的基本原理；然后，我会带你进入 Graal 编译器的代码内部，一起去修改它、运行它、调试它，让你获得第一手的实践经验，消除你对 JIT 编译器的神秘感。

认识 Java 的 JIT 编译器

我们先来探究一下 JIT 编译器的原理。

在第 5 讲中，我讲过程序运行的原理：把一个指令指针指向一个内存地址，CPU 就可以读取其中的内容，并作为指令来执行。

所以，Java 后端的编译器只要生成机器码就行了。如果是在运行前一次性生成，就叫做提前编译（AOT）；如果是在运行时按需生成机器码，就叫做即时编译（JIT）。Java 以及基于 JVM 的语言，都受益于 JVM 的 JIT 编译器。

在 JDK 的源代码中，你能找到 src/hotspot 目录，这是 JVM 的运行时，它们都是用 C++ 编写的，其中就包括 JIT 编译器。标准 JDK 中的虚拟机呢，就叫做 HotSpot。

实际上，HotSpot 带了两个 JIT 编译器，一个叫做 **C1**，又叫做**客户端编译器**，它的编译速度快，但优化程度低。另一个叫做 **C2**，又叫做**服务端编译器**，它的编译速度比较慢，但优化程度更高。这两个编译器在实际的编译过程中，是被结合起来使用的。而**字节码解释器**，我们可以叫做是 **C0**，它的运行速度是最慢的。

在运行过程中，HotSpot 首先会用 C0 解释执行；接着，HotSpot 会用 C1 快速编译，生成机器码，从而让运行效率提升。而对于运行频率高的热点（HotSpot）代码，则用 C2 深化编译，得到运行效率更高的代码，这叫做**分层编译**（Tiered Compilation）。

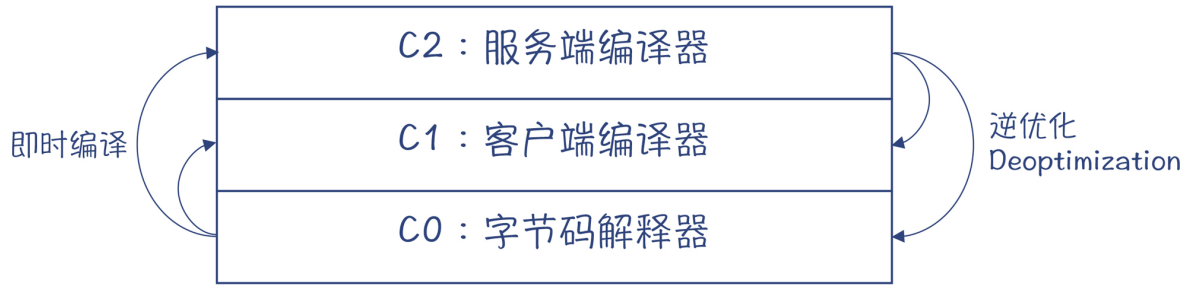


图 1：分层编译

由于 C2 会做一些激进优化，比如说，它会根据程序运行的统计信息，认为某些程序分支根本不会被执行，从而根本不为这个分支生成代码。不过，有时做出这种激进优化的假设其实并不成立，那这个时候就要做一个**逆优化（Deoptimization）**，退回到使用 C1 的代码，或退回到用解释器执行。

触发即时编译，需要检测**热点代码**。一般是以方法为单位，虚拟机会看看该方法的运行频次是否很高，如果运行特别频繁，那么就会被认定为是热点代码，从而就会被触发即时编译。甚至如果一个方法里，有一个循环块是热点代码（比如循环 1.5 万次以上），这个时候也会触发编译器去做即时编译，在这个方法还没运行完毕的时候，就被替换成了机器码的版本。由于这个时候，该方法的栈帧还在栈上，所以我们把这个技术叫做**栈上替换**（On-stack Replacement，OSR）。栈上替换的技术难点，在于让本地变量等数据无缝地迁移，让运行过程可以正确地衔接。

Graal：用 Java 编写的 JIT 编译器

如果想深入地研究 Java 所采用的 JIT 编译技术，我们必须去看它的源码。可是，对于大多数 Java 程序员来说，如果去阅读 C++ 编写的编译器代码，肯定会有些不适应。

一个好消息是，Oracle 公司推出了一个完全用 Java 语言编写的 JIT 编译器：Graal，并且也有开放源代码的社区版，你可以下载安装并使用。

用 Java 开发一款编译器的优点是很明显的。

1. 首先，Java 是内存安全的，而 C++ 程序的很多 Bug 都与内存管理有关，比如可能不当地使用了指针之类的。

2. 第二，与 Java 配套的各种工具（比如 IDE）更友好、更丰富。
3. 第三，Java 的性能并不低，所以能够满足对编译速度的需求。
4. 最后，用 Java 编译甚至还能节省内存的占用，因为 Java 采用的是动态内存管理技术，一些对象没用了，其内存就会被回收。而用 C++ 编写的话，可能会由于程序员的疏忽，导致一些内存没有被及时释放。

从 Java9 开始，你就可以用 Graal 来替换 JDK 中的 JIT 编译器。这里有一个 **JVMCI**（JVM Compiler Interface）接口标准，符合这个接口标准的 JIT 编译器，都可以被用于 JVM。

Oracle 公司还专门推出了一款 JVM，叫做 **GraalVM**。它除了用 Graal 作为即时编译器以外，还提供了一个很创新的功能：在一个虚拟机上支持多种语言，并且支持它们之间的互操作。你知道，传统的 JVM 上已经能够支持多种语言，比如 Scala、Clojure 等。而新的 GraalVM 会更进一步，它通过一个 Truffle 框架，可以支持 JavaScript、Ruby、R、Python 等需要解释执行的语言。

再进一步，它还通过一个 Sulong 框架支持 LLVM IR，从而支持那些能够生成 LLVM IR 的语言，如 C、C++、Rust 等。想想看，在 Java 的虚拟机上运行 C 语言，还是有点开脑洞的！

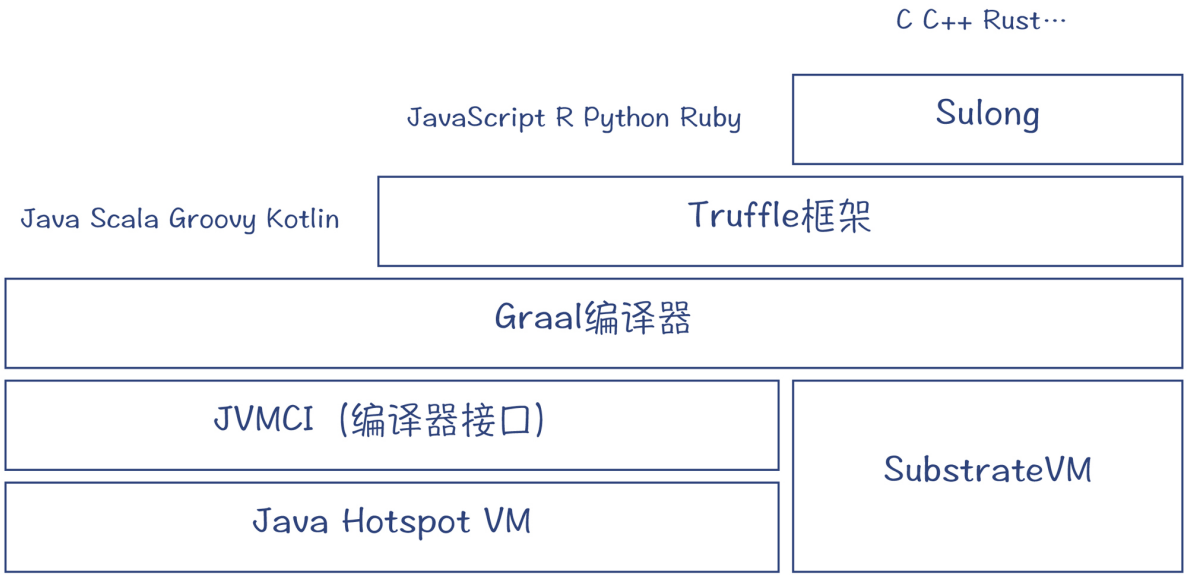


图 2：GraalVM 的架构

最后，GraalVM 还支持 AOT 编译，这就让 Java 可以编译成本地代码，让程序能更快地启动并投入高速运行。我听说最近的一些互联网公司，已经在用 Graal 做 AOT 编译，来生成本地镜像，提高应用的启动时间，从而能够更好地符合云原生技术的要求。

修改并运行 Graal

好，那接下来，我就带你一起动手修改一下 Graal 编译器，在这个过程中，你就能对 Graal 的程序结构熟悉起来，消除对它的陌生感，有助于后面深入探索其内部的实现机制。

在本课程中，我采用了 Graal 的 20.0.1 版本的源代码。你可以参考 Graal 中的文档来做编译工作。

首先，下载源代码（指定了代码的分支）：

复制代码

```
1 git clone -b vm-20.0.1 https://github.com/oracle/graal.git
```

接着，下载 GraalVM 的构建工具 mx，它是用 Python2.7 编写的，你需要有正确的 Python 环境：

复制代码

```
1 git clone https://github.com/graalvm/mx.git
2 export PATH=$PWD/mx:$PATH
```

你需要在自己的机器上设置好 JDK8 或 11 的环境。我这里是在 macOS 上，采用 JDK8。

复制代码

```
1 export PATH="/Library/Java/JavaVirtualMachines/openjdk1.8.0_252-jvmci-20.1-b02
2 export JAVA_HOME=/Library/Java/JavaVirtualMachines/openjdk1.8.0_252-jvmci-20.1
```

好了，现在你就可以编译 Graal 了。你可以在 Graal 源代码的 compiler 子目录中，运行 mx build：

```
mx build
```

复制代码

编译完毕以后，你可以写一个小小的测试程序，来测试 Graal 编译器的功能。

复制代码

```
javac Foo.java    //编译Foo.java
mx vm Foo         //运行Foo.java，相当于执行java Foo
```

“mx vm”命令在第一次运行的时候，会打包出一个新的 GraalVM，它所需要的 HotSpot VM，是从 JDK 中拷贝过来的，然后它会把 Graal 编译器等其他模块也添加进去。

Foo.java 的源代码如下。在这个示例程序中，main 方法会无限次地调用 add 方法，所以 add 方法就成为了热点代码，这样会逼迫 JIT 编译器把 add 方法做即时编译。

复制代码

```
public class Foo{
    public static void main(String args[]){
        int i = 0;
        while(true){
           if(i%1000==0){
                System.out.println(i);
                try{
                    Thread.sleep(100);  //暂停100ms
                }catch(Exception e){}
            }

            i++;
            add(i,i+1);
        }
    }


    public static int add(int x, int y){
        return x + y;
    }
```

由于我们现在已经有了 Graal 的源代码，所以我们可以在源代码中打印一点信息，来显示 JIT 是什么时候被触发的。

org.graalvm.compiler.hotspot.HotspotGraalCompiler.compileMethod() 方法，是即时编译功能的入口，你可以在里面添加一行输出功能，然后用“mx build”命令重新构建。

复制代码

```
public CompilationRequestResult compileMethod(CompilationRequest request) {
    //打印被编译的方法名和字节码
    System.out.println("Begin to compile method: " + request.getMethod().getNa

    return compileMethod(request, true, graalRuntime.getOptions());
}
```

你在 compiler 目录里，打出“mx ideinit”命令，就可以为 Eclipse、IntelliJ Idea 等编译器生成配置信息了。你可以参照文档来做好 IDE 的配置。

注意：我用 Eclipse 和 IntelliJ Idea 都试了一下。Idea 的使用体验更好一些。但用 mx ideinit 命令为 Idea 生成的配置文件，只是针对 JDK8 的，如果要改为 JDK11，还需要手工修改不少配置信息。

在使用 Idea 的时候，你要注意安装 python 插件，文档中建议的其他插件可装可不装。

在使用 Eclipse 时，我曾经发现有一些报错信息，是因为 IDE 不能理解一些注解。你如果也遇到了类似情况，稍微修改一下头注释就能正常使用了。

复制代码

```
mx ideinit
```

然后，你可以运行下面的命令来执行示例程序：

复制代码

```
mx vm \
  -XX:+UnlockExperimentalVMOptions \
  -XX:+EnableJVMCI \
  -XX:+UseJVMCICompiler \
  -XX:-TieredCompilation \
  -XX:CompileOnly=Foo.add \
Foo
```

你会看到，命令中包含了很多不同的参数，它们分别代表了不同的含义。

-XX:+UnlockExperimentalVMOptions：启用试验特性。

-XX:+EnableJVMCI：启用 JVMCI 功能。

-XX:+UseJVMCICompiler ：使用 JVMCI 编译器，也就是 Graal。

-XX:-TieredCompilation ：禁用分层编译。

-XX:CompileOnly=Foo.add：只编译 add 方法就行了。

当程序运行以后，根据打印的信息，你就能判断出 JIT 编译器是否真的被调用了。实际上，它是在 add 方法执行了 15000 次以后才被调用的。这个时候，JVM 会认为 add 方法是一个热点。因为 JIT 是在另一个线程启动执行的，所以输出信息要晚一点。

好了，通过这个实验，你就能直观地了解到，JVM 是如何判断热点并启动 JIT 机制的了。

```
Java HotSpot(TM) 64-Bit Server VM warning: -XX:ThreadPriorityPolicy=1 may require sy
stem level permission, e.g., being the root user. If the necessary permission is not
 possessed, changes to priority will be silently ignored.
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
Begin to compile method: add
bytecode: [26, 27, 96, -84]
19000
```

另外，在这个实验中，你还可以通过“-XX:CompileThreshold”参数，来修改热点检测的门槛。比如说，你可以在“-XX:CompileThreshold=50”，也就是让 JVM 在被 add 方法执行了 50 次之后，就开始做即时编译。你还可以使用“-Xcomp”参数，让方法在第一次被调用的时候就开始做编译。不过这样编译的效果会差一些，因为让方法多运行一段时间

再编译，JVM 会收集一些运行时的信息，这些信息会有助于更好地做代码优化。**这也是 AOT 编译的效果有时会比 JIT 差的原因，因为 AOT 缺少了运行时的一些信息。**

好了，接下来，我们再来看看 JIT 编译后的机器码是什么样子的。

JIT 所做的工作，本质上就是把字节码的 Byte 数组翻译成机器码的 Byte 数组，在翻译过程中，编译器要参考一些元数据信息（符号表等），再加上运行时收集的一些信息（用于帮助做优化）。

前面的这个示例程序，它在运行时就已经打印出了字节码：[26, 27, 96, -84]。如果我们转换成 16 进制，就是[1a, 1b, 60, ac]。它对应的字节码是：[iload_0, iload_1, iadd, ireturn]。

我们暂时忽略掉这中间的编译过程，先来看看 JIT 编译后生成的机器码。

Graal 编译完毕以后，是在 org.graalvm.compiler.hotspot.CompilationTask 的 **performCompilation 方法**中，把编译完毕的机器码安装到缓存区，用于后续执行。在这里，你可以加一点代码，打印编译后的结果。

复制代码

```
...
installMethod(debug, result);   //result是编译结果
System.out.println("Machine code: " + java.util.Arrays.toString(result.getTarg
...
```

打印输出的机器码数组如下：

```
Machine code: [15, 31, 68, 0, 0, 3, -14, -117, -58, 73, -117, -113, 8, 1, 0, 0, -123
, 1, -61, -24, 0, 0, 0, 0, -112, -24, 0, 0, 0, 0, -112, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

运行“mx hsdis”命令，你可以下载一个动态库（在 macOS 上是 hsdis-amd64.dylib，在 Linux 上以 so 结尾，在 Windows 上以 dll 结尾）。这个动态库会被拷贝到 JDK 的 lib 目录下，这样我们就可以通过命令行参数，让 JVM 输出编译生成的汇编码。

复制代码

```
sudo -E mx hsdis #用sudo是为了有权限把动态库拷贝到系统的JDK的lib目录
```

注：由于我使用 mx 命令来运行示例程序，所以使用的 JDK 实际上是 GraalVM 从系统 JDK 中拷贝过来的版本，因此我需要手工把 hsdis.dylib 拷贝到 graal-vm-20.0.1/compiler/mxbuild/darwin-amd64/graaljdks/jdk11-cmp/lib 目录下。

复制代码

```
mx vm \
  -XX:+UnlockExperimentalVMOptions \
  -XX:+EnableJVMCI \
  -XX:+UseJVMCICompiler \
  -XX:-TieredCompilation \
  -XX:+PrintCompilation \
  -XX:+UnlockDiagnosticVMOptions \
  -XX:+PrintAssembly \
  -XX:CompileOnly=Foo.add \
Foo
```

输出的汇编码信息如下：

```
Foo.add(II)I  [0x0000000101855880, 0x00000001018558a0]  32 bytes
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} {0x0000000115cb5478} 'add' '(II)I' in 'Foo'
  # parm0:    rsi       = int
  # parm1:    rdx       = int
  #           [sp+0x10]  (sp of caller)
  0x0000000101855880: nopl   0x0(%rax,%rax,1)
  0x0000000101855885: add    %edx,%esi          ;*iadd {reexecute=0 rethrow=0 return_oop=0}
                                                ; - Foo::add@2 (line 24)

  0x0000000101855887: mov    %esi,%eax          ;*ireturn {reexecute=0 rethrow=0 return_oop=0}
                                                ; - Foo::add@3 (line 24)

  0x0000000101855889: mov    0x108(%r15),%rcx
  0x0000000101855890: test   %eax,(%rcx)        ;   {poll_return}
  0x0000000101855892: retq
[Exception Handler]
  0x0000000101855893: callq  0x0000000101855580  ;   {runtime_call Stub<ExceptionHandlerStub.exceptionHandler>}
  0x0000000101855898: nop
[Deopt Handler Code]
  0x0000000101855899: callq  0x00000001017eb6a0  ;   {runtime_call DeoptimizationBlob}
  0x000000010185589e: nop
[Stub Code]
  0x000000010185589f: hlt
```

我来解释一下这段汇编代码的含义：

nopl 0x0(%rax,%rax,1)	不做任何操作
add %edx,%esi	把%edx的值加到%esi上 根据调用约定，%esi存放的是第二个参数，%edx存放的是第三个参数，这实际上完成的是a+b的运算
mov %esi,%eax	把结果放到%eax寄存器 根据调用约定，这个寄存器是用来保存方法的返回值
mov 0x108(%r15),%rcx	这两行是HotSpot的安全点机制，让虚拟机可以在这个点做与GC有关的操作
test %eax,(%rcx)	
retq	正常返回
callq 0x0000000101855580	执行异常处理程序
nop	
callq 0x00000001017eb6a0	执行逆优化处理程序
nop	
hlt	

好了，现在你已经能够直观地了解 JIT 启动的时机：检测出热点代码；以及它最后生成的结果：机器码。

但我们还想了解一下中间处理过程的细节，因为这样才能理解编译器的工作机制。所以这个时候，如果能够跟踪 Graal 的执行过程就好了，就像调试一个我们自己编写的程序那样。那么我们能做到吗？

当然是可以的。

跟踪 Graal 的运行

Graal 是用 Java 编写的，因此你也可以像调试普通程序一样调试它。你可以参考源代码中的这篇与调试有关的文档。

由于 Graal 是在 JVM 中运行的，所以你要用到 JVM 的远程调试模式。我们仍然要运行 Foo 示例程序，不过要加个“-d”参数，表示让 JVM 运行在调试状态下。

复制代码

```
mx -d vm \
  -XX:+UnlockExperimentalVMOptions \
  -XX:+EnableJVMCI \
  -XX:+UseJVMCICompiler \
  -XX:-TieredCompilation \
  -XX:CompileOnly=Foo.add \
Foo
```

这个时候，在 JVM 启动起来之后，会在 8000 端口等待调试工具跟它连接。

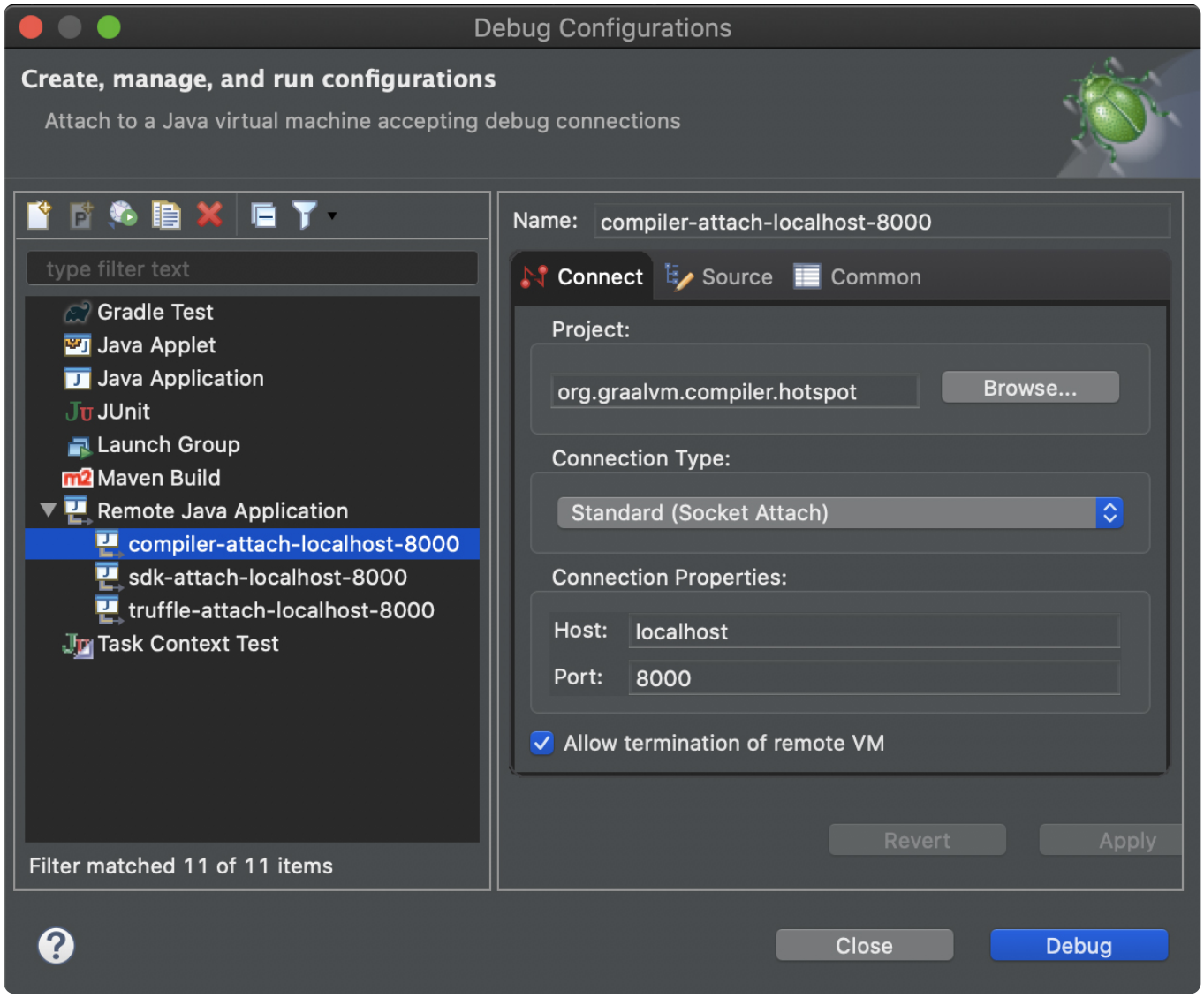
```
Listening for transport dt_socket at address: 8000
```

你可以使用 Eclipse 或 Idea 做调试工具。我以 Eclipse 为例，在前面运行“mx ideinit”的时候，我就已经设置了一个远程调试的配置信息。

你可以打开“run>debug configurations...”菜单，在弹出的对话框中，选择 Remote Java Application，可以看到几个预制好的配置。

然后，点击“compiler-attach-localhost-8000”，你可以看到相关属性。其中，连接信息正是本机的 8000 端口。

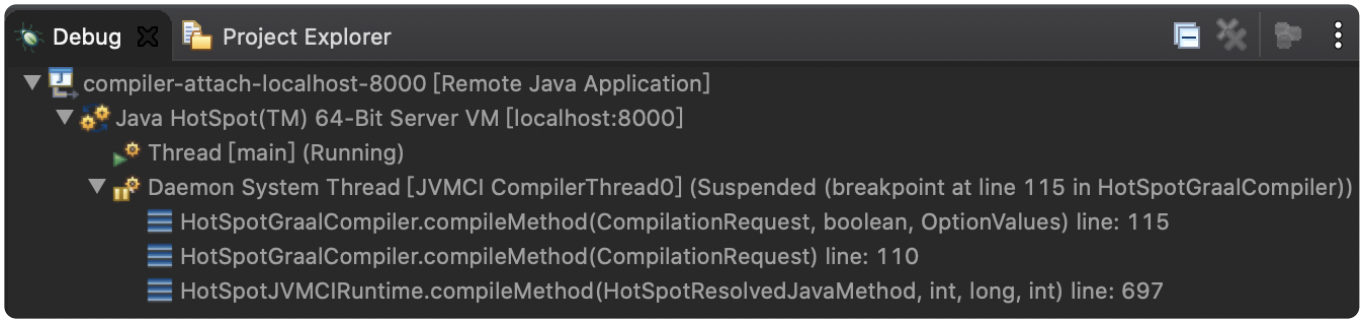
把 Project 改成“org.graalvm.compiler.hotspot”，然后点击 Debug 按钮。



补充：如果你使用的是 Idea，你也会找到一个预制好的远程调试配置项：GraalDebug。直接点击就可以开始调试。

为了方便调试，我在 org.graalvm.compiler.hotspot.compileMethod() 方法中设置了断点，所以到了断点的时候，程序就会停下来，而不是一直运行到结束。

当你点击 Debug 按钮以后，Foo 程序会继续运行。在触发了 JIT 功能以后，JVM 会启动一个新线程来运行 Graal，而 Foo 则继续在主线程里运行。因为 Foo 一直不会结束，所以你可以从容地进行调试，不用担心由于主线程的退出，而导致运行 Graal 的线程也退出。



现在，你可以跟踪 Graal 的编译过程，看看能发现些什么。在这个过程中，你需要一点耐心，慢慢理解整个代码结构。

Graal 执行过程的主要结构如下图所示。

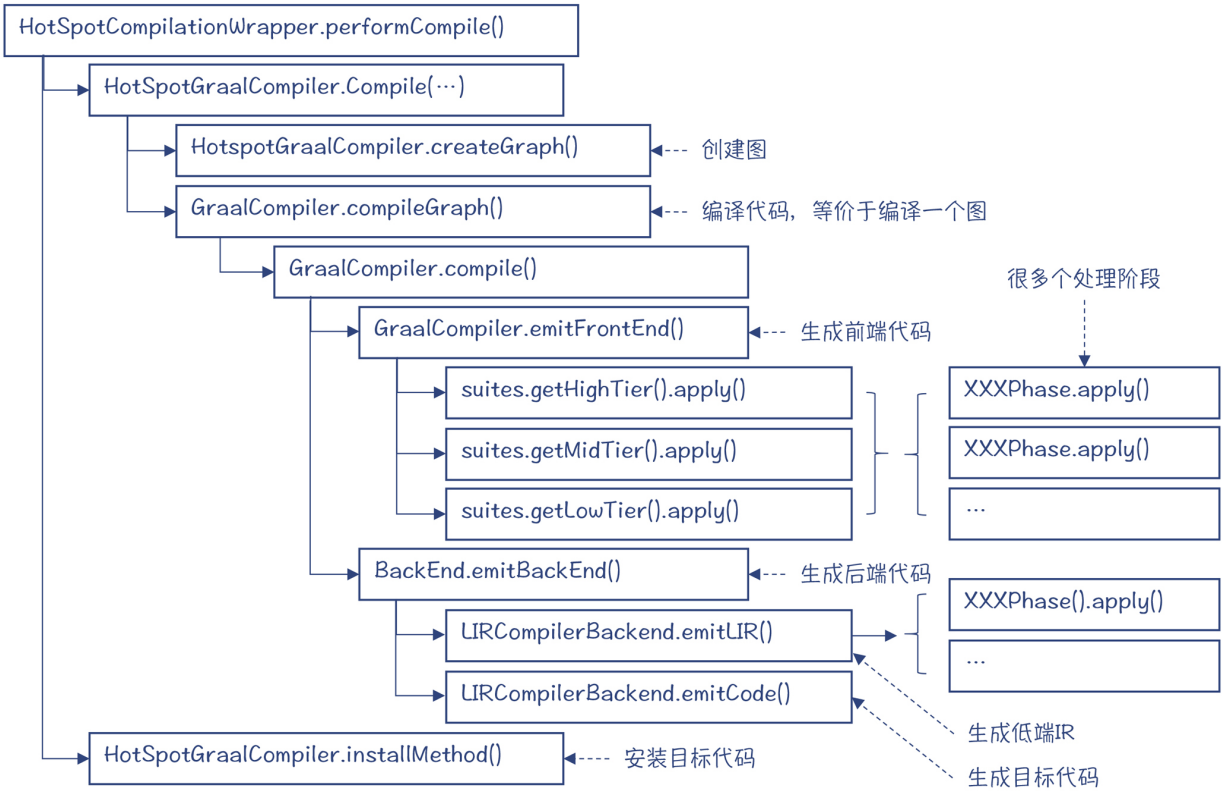


图 3：Graal 执行过程的主要结构

首先，你会发现，在编译开始的时候，Graal 编译器要把字节码转化成一个图的数据结构。而后续的编译过程，都是对这个图的处理。这说明了这个图很重要，而这个图就是 Graal 要用到的 IR，在 Graal 编译器中，它也被叫做 HIR。

接着，你会看到，整个编译过程被分成了前端和后端两个部分。前端部分使用的 IR 是 HIR。而且在前端部分，HIR 又分成了高（HighTier）、中（MidTier）、低（LowTier）

三层。在每个层次里，都要执行很多遍（Phase）对图的处理。这些处理，指的就是各种的优化和处理算法。而从高到低过渡的过程，就是不断 Lower 的过程，也就是把 IR 中，较高抽象度的节点替换成了更靠近底层实现的节点。

在后端部分，则要生成一种新的 IR，也就是我们在第 6 讲中提到过的 LIR，并且 Graal 也要对它进行多遍处理。最后一步，就是生成目标代码。

下图中，我举了一个例子，列出了编译器在前端的三个层次以及在后端所做的优化和处理工作。

你要注意的是，在编译不同的方法时，所需要的优化工作也是不同的，具体执行的处理也就不一样。并且这些处理执行过程也不是线性执行的，而可能是一个处理程序调用了另一个处理程序，嵌套执行的。

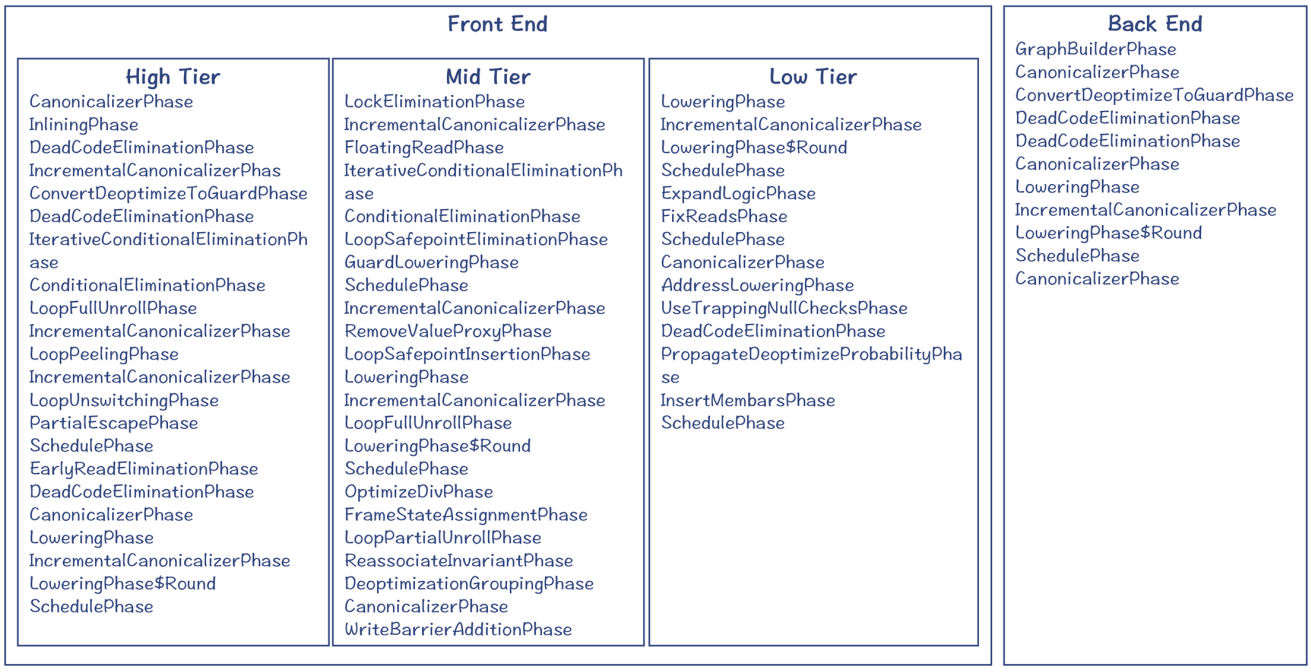


图 4：一个例子，前端三个层次和后端所做的处理

不过通过跟踪 Graal 的运行过程，你可以留下一个直观的印象：**Graal 编译器的核心工作，就是对图（IR）的一遍遍处理。**

在下一讲中，我就会进一步讲述 Graal 的 IR，并会带你一起探讨优化算法的实现过程，你可以了解到一个真实编译器的 IR 是怎样设计的。

课程小结

今天这一讲，我带你大致了解了 Java 的 JIT 编译器。你需要重点关注以下几个核心要点：

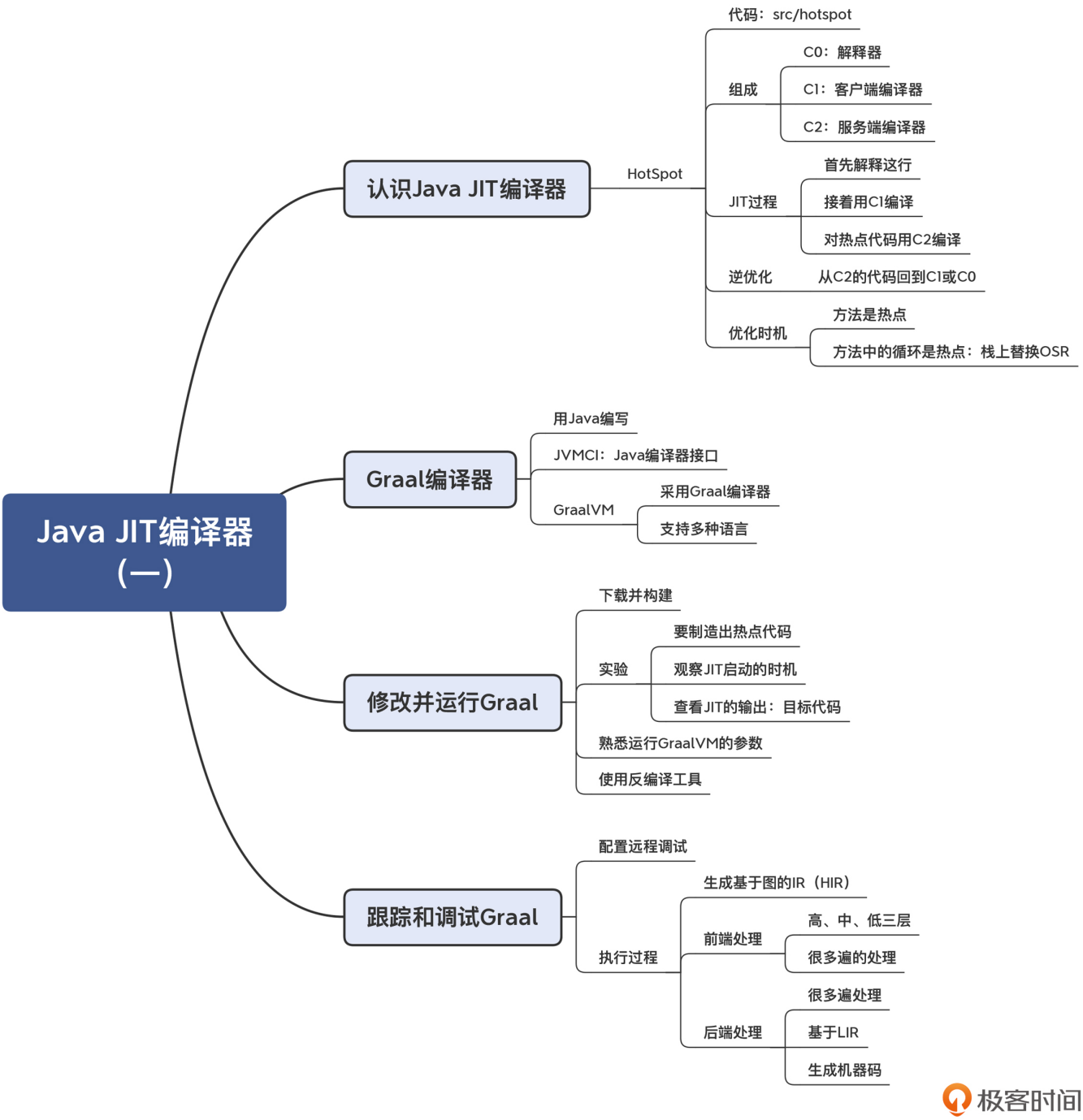
JIT 可能会用到多个编译器，有的编译速度快但不够优化（比如 C1，客户端编译器），有的够优化但编译速度慢（比如 C2，服务端编译器），所以在编译过程中会结合起来使用。

你还需要理解逆优化的概念，以及导致逆优化的原因。

另外，我还带你了解了 Graal 这个用 Java 编写的 Java JIT 编译器。最重要的是，通过查看它的代码、修改代码、运行和调试的过程，你能够建立起对 Graal 编译器的亲切感，不会觉得这些技术都是很高冷的，不可接近的。当你开始动手修改的时候，你就踏上了彻底掌握它的旅途。

你要熟练掌握调试方法，并且熟练运用 GraalVM 的很多参数，这会有利于你去做很多实验，来深入掌握 Graal。

本讲的思维导图我放在这里了，供你参考：



一课一思

你能否把示例程序的 add 函数改成一个需要计算量的函数，然后，你可以比较一下，看看 JIT 前后性能相差了多少倍？通过这样的一个例子，你可以获得一些感性认识。

有相关的问题或者是思考呢，你都可以给我留言。如果你觉得有收获，你也可以把今天的内容分享给更多的朋友。

参考资料

1. GraalVM 项目的官方网站：graalvm.org。

2. Graal 的Github 地址。

3. Graal 项目的出版物。有很多围绕这个项目来做研究的论文，值得一读。

提建议

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | Java编译器（四）：去除语法糖和生成字节码

下一篇 14 | Java JIT编译器（二）：Sea of Nodes为何如此强大？

精选留言 (4)

写留言

wusiration
2020-07-04

这是目前在win 7环境下的编译报错，
Extracting LIBFFI_SOURCES...
Applying patches...
error: invalid path 'truffle\mxbuild\windows-amd64\src\libffi\libffi-3.2.1/src/x86/ffi.c...

展开

作者回复: 在windows上，你直接试试release版本怎么样？参见：
https://www.graalvm.org/docs/getting-started/windows

如果解决不了，你继续留言，我找个Windows环境测试一下。

其他同学遇到类似问题，也可以交流下。

1

l6
2020-07-07

想知道老师的结构图一类的是什么工具画的

展开

作者回复: PowerPoint :-)

wusiration
2020-07-01

老师，windows是不是很难编译graal，用mx工具build的时候，不停报错...

作者回复: 理论上，在不同平台上编译graal是一样的，因为是纯Java的嘛。你注意把JDK、Python等环境配置好就行了。
你也可以把错误贴上来，我看一下。

1

Jxin
2020-07-01

本来我觉得jdk8的升级应该会很缓慢。但jdk9的aop编译如果不以补丁包加到jdk8，怕是能有效加快这个过程。毕竟原本不是很重要的启动时间，在servless下，变得至关重要。

作者回复: 是。Servless，云原生，让启动时间变得敏感。